Variables dans Python

1 - Règles de nommage

1.1 minuscule initiale

Un nom qui commence par une majuscule, c'est pour autre chose.

1.2 Pas d'espace

En effet, l'espace joue le rôle de délimitateur. Par contre, **Python est sensible à la casse** : il distingue les minuscules et les majuscules. Les variables `toto` et `toTO` désignent deux variables différentes.

Pour les longs noms de variables, il existe 2 techniques.

snake_case : utiliser des underscores entre les mots.

— ⇒ `position_voiture`
— ⇒ `liste_ennemis`

CamelCase : utiliser une majuscule pour séparer les mots.

— ⇒ `positionVoiture`
— ⇒ `listeEnnemis`

1.3 Nom explicite

Donnez à vos variables des **noms explicites** permettant de savoir au premier coup d'oeil ce qu'elles contiennent.

N'utilisez jamais de variable `l` (L minuscule) : trop difficile à distinguer d'un `I` ou d'un `1`.

1.4 Noms réservés

Voir le site pour la liste des mots non utilisables.

1.5 Caractères spéciaux

Python accepte n'importe quel caractère dans les noms de variables (**à part l'espace**). Néanmoins, n'utilisez que les caractères non accentués.

2 - Utilisation de variables

2.1 Affectation (rappel)

2.2 Incrémentation(rappel)

Voir Python Introduction 1.4.2 et 1.4.3

2.3 Vocabulaire

Expression

Une expression est un ensemble de valeurs associées à des opérateurs. L'interpréteur peut évaluer l'expression pour en fournir la valeur.

Une expression **ne modifie pas l'état du programme**. Elle fournit simplement une valeur, sans la stocker définitivement.

```
1    "bon" * 2  # programme
```

Ce programme n'a aucun effet, ni en mémoire, ni à l'affichage.

```
>>> "bon" * 2  # console
'bonbon'
```

Par contre, notez bien que **la console Python affiche le résultat d'une expression non redirigée ailleurs**.

Instruction

Une instruction **modifie l'état du programme**. L'instruction la plus fondamentale est l'**affectation** : on crée une variable.

```
1    a = 5
```

L'utilisation d'une méthode du module `Turtle` est bien une instruction : on modifie l'affichage de l'interface graphique.

```
>>> "bon" * 2  # Exp.
'bonbon'

>>> a = "bon" * 2  # Inst. et aff.
```

```
>>> print("bon" * 2)  # Inst.
  bonbon
```

2.4 Pas de retroaction

Une variable n'est jamais recalculée automatiquement.

```
>>> a = 10
>>> b = a
>>> a = 20
>>> b
10  # b non re-modifiée
```

2.5 Permutation de deux contenus

Méthode en 3 lignes avec Python

Il faut une troisième variable.

```
>>> g = 7
>>> d = 55
>>> videPoche = d
>>> d = g
```

```
>>> g = videPoche
>>> g     >>> d
55        7
```

Méthode en une ligne

Facilitateur à éviter en NSI.

```
>>> g = 7
>>> d = 55
>>> g, d = d, g
```

2.6 Variable,CONSTANTE

Définition : variable

Une variable est un symbole qui associe un nom à une valeur. L'association est **variable au cours du temps**.

```
1   a = 10
2   ...
3   a = 20
```

Définition : CONSTANTE

Une CONSTANTE est un symbole qui associe DEFINITIVEMENT un nom à une valeur. Par convention, les noms des constantes sont constitués de majuscules.

```
1   NBR_CASES = 10
2   ...
```

En Python

Il n'y a aucun moyen de créer de véritables constantes en Python.

2.7 Affectations multiples

Un autre facilicitateur (à éviter en NSI) :

```
>>> a = b = c = 10
>>> a    >>> b    >>> c
10       10       10
```

3 - Documenter le type

3.1 DOCUMENTATION du type

Principe en Python

En Python, il est possible (pas obligatoire) d'indiquer, autrement qu'avec un commentaire #, le type de telle ou telle variable. Lorsqu'on déclare une variable, on peut intercaler le type avant de placer l'opérateur d'affectation =.

```
1   t1 :list = [5, 10, 15]
2   t2 :list = [10, 20]
3   texte :str = "Bon"
4   t3 :list = t1 + t2
```

Sur les lignes 1 à 3, on voit bien que l'indication ne sert à rien : on affecte directement à droite un tableau ou un string bien visible. Par contre, c'est un peu plus utile sur la deuxième ligne d'exemple.

Sans utilisation d'un module supplémentaire, cela n'est qu'une simple indication, un commentaire pour la personne qui va lire votre code. Nous pourrions faire la même chose que ci-dessous en plaçant juste des # indiquant le type sur chaque ligne :

```
1   texte :str = "Bon"
2
3   print("Combien de fois ? ")
4   nombre :int = int( input() )
5
6   resultat :str = texte * nombre
7   print(resultat)
```

Principe en Python

En Python, cette façon d'indiquer les types n'est pas souvent utilisés car elle n'était pas présente dans les premières versions et la plupart du temps le code lui-même est déjà explicite quand à la nature des données référencées.

3.2 DOCUMENTATION rapide des fonctions

Signature d'une fonction

Pour pouvoir utiliser une fonction, on a besoin de savoir 3 choses : comment elle se nomme, combien d'arguments lui envoyer et leurs types, ce qu'elle va répondre.

Toute ces informations se retrouvent dans la signature. Par exemple, la signature d'une fonction **`addition()`** qui permet d'additionner deux entiers :

```
addition(int, int) -> int
```

Prototype d'une fonction

Prototyper une fonction consiste à rajouter les noms des paramètres à la signature.

```
addition(nombre1:int, nombre2:int) -> int
```

Spécification

Il s'agit de la documentation minimale d'une fonction. Spécifier une fonction consiste à donner :

— Le PROTOTYPE qui fixe la SYNTAXE et le TYPAGE

— Une description nommée DOCSTRING, délimitée par trois guillemets d'ouverture et de fermeture, qui fixe clairement la SEMANTIQUE.

```python
def addition(nbr1 :int, nbr2 :int) -> int :
    """Renvoie la somme de nbr1 et nbr2"""
    return nbr1 + nbr2
```

Obtenir la documentation

La fonction `help()` renvoie la documentation.

```
>>> help(addition)
Help on function addition in module __main__ :

addition(nbr1 : int, nbr2 : int) -> int
    Renvoie la somme de nbr1 et nbr2
```

3.3 Gestion du typage

Python est un langage utilisant un **typage dynamique** : le type d'une variable est décidé par l'intepréteur au moment de son affectation.

D'autres langages, comme le C, adopte le **typage statique** : le développeur doit choisir définitivement le type associé à sa variable.

D'autres langages, comme Ocaml, ne nécessite pas nécessairement le type des variables mais c'est le langage lui-même qui infère (déduit) des différents types en fonction des lignes qu'il analyse.

4 - Portée des variables

4.1 Variable locale

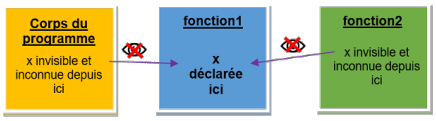

FIGURE 6.1 – portée variable locale

Variable locale : variable temporaire définie dans une fonction.

- Avant l'appel, cette variable n'existe pas.
- Pendant l'appel, elle existe pendant **cet appel précis**.
- Après le `return`, cette variable est détruite définitivement.

Conséquences :

- Depuis **cet appel** : la variable est accessible **en lecture et modification**.
- Depuis le **programme principal** : elle est inconnue.
- Depuis une **autre fonction** : elle est inconnue.
- Depuis un **autre appel de la même fonction** : la variable du nouvel appel porte le même nom, mais il s'agit bien d'une autre variable.

Exemple d'illustration :

```python
def f(valeur) :
    a = valeur * 2
    print("Depuis f :")
    print(a)  # Ok

print(a)  # Erreur
f(10)
print(a)  # Erreur
```

4.2 Variable globale

FIGURE 6.2 – portée variable globale

Variable globale : une variable **permanente définie dans le programme hors de toute fonction**. Permanente, elle existe jusqu'à la fin du programme.

- Depuis le **prog. principal** : elle est accessible **en lecture et modification**.
- Depuis les **fonctions** : elle est accessible en **lecture uniquement** : une nouvelle affectation avec ce nom crée simplement une variable locale de même nom.

Seule possibilité pour qu'elle disparaisse : la supprimer volontairement avec le mot-clé `del`. Exemple : `del a` permet de supprimer la variable a.

Exemple de lecture possible : en ligne 2, on lit `valeur` directement depuis la fonction.

```python
def f() :
    a = valeur * 2
    print("Depuis f")
    print(a)

valeur = 10
f()
```

```
Depuis f    -> 20
```

Modification impossible En ligne 2, on crée en réalité une variable locale qui disparaitra après l'appel.

```
1   def f() :
2       valeur = 100
3       print("Depuis f")
4       print(valeur)
5
6   valeur = 2
7   print("Avant f")
8   print(valeur)
9   f()
10  print("Après f")
11  print(valeur)
```

```
Avant f    ->  2
Depuis f   ->  100
Après f    ->  2
```

4.3 Savoir différencier locales et globales

Exemple

```
1   def modification(a) :    # a locale
2       print(a)             # a locale
3       a = 5                # a locale
4       print(a)             # a locale
5       return a * 10        # a locale
6
7   a = 10                   # a globale
8   b = modification(a)      # a globale
9   print(a)                 # a globale
```

— **Locales** : `a`
— **Globales** : `modification()`, `a`, `b`

Comment ne pas s'embrouiller ?

On rajoute mentalement des suffixes aux variables qui portent le même nom ! Exemple :

— `a_mod` pour les `a` dans la fonction `modification()`.
— `a_glo` pour les `a` dans le programme principal.

```
1   def modification(a_mod) :
2       print(a_mod)
3       a_mod = 5
4       print(a_mod)
5       return a_mod * 10
6
7   a_glo = 10
8   b = modification(a_glo)
9   print(a_glo)
```

L1-L5, `a_mod` est locale. L7-L9, `a_glo` est globale.

Au début de l'appel, puisqu'on envoie `a_glo` dans `a_loc`, ce sont temporairement des alias. En ligne 3, affectation, elle devient locale.

4.4 Variables : Espace des noms

Les espaces des noms

Lors d'une affectation, on crée une association entre un nom de variable et un contenu-mémoire. L'ensemble de ces **associations** se nomme l'**espace des noms**. Il existe **plusieurs espaces des noms**.

— L'**espace des noms global permanent** du programme.
— Chaque appel de fonction crée son propre **espace des noms local temporaire**, détruit après l'appel.

Evaluation d'une variable par Python

— il cherche d'abord dans l'**espace local** des noms.
— ensuite, il cherche dans l'**espace global** des noms.
— Sans résultat, il déclenche une `exception NameError`.

Exemple :

```
1   def f(x) :
2       return x * 2
3
4   a = 10
5   b = f(a)
```

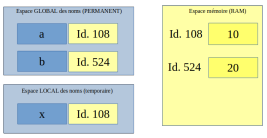

FIGURE 6.3 – portée variable locale

4.5 `randint()` du module random

La fonction `randint()` génère un integer aléatoire entre des valeurs incluses. Un programme générant 5 notes entre 0 et 20 :

```
1   import random  # Importation du module
2   notes = [0, 0, 0, 0, 0]
3   for i in range(5) :
4       notes[i] = random.randint(0, 20)
5   print(notes)
```

```
[19, 14, 5, 7, 3]
```